33 | 编译原理（中）：Vue Compiler模块全解析

2022-01-05 大圣

《玩转Vue 3全家桶》 课程介绍 >

讲述：大圣

时长 08:46 大小 8.04M

你好，我是大圣。

上一讲我带你手写了一个迷你的 Vue compiler，还学习了编译原理的基础知识。通过实现这个迷你 Vue compiler，我们知道了 tokenizer 可以用来做语句分析，而 parse 负责生成抽象语法树 AST。然后我们一起分析 AST 中的 Vue 语法，最后通过 generate 函数生成最终的代码。

今天我就带你深入 Vue 的 compiler 源码之中，看看 Vue 内部到底是怎么实现的。有了上一讲编译原理的入门基础，你会对 Compiler 执行全流程有更深的理解。

Vue compiler 入口分析

Vue 3 内部有 4 个和 compiler 相关的包。compiler-dom 和 compiler-core 负责实现浏览器端的编译，这两个包是我们需要深入研究的，compiler-ssr 负责服务器端渲染，我们后面讲 ssr 的时候再研究，compiler-sfc 是编译.vue 单文件组件的，有兴趣的同学可以自行探索。

首先我们进入到 vue-next/packages/compiler-dom/index.ts 文件下，在GitHub上你可以找到下面这段代码。

compiler 函数有两个参数，第一个参数 template，它是我们项目中的模板字符串；第二个参数 options 是编译的配置，内部调用了 baseCompile 函数。我们可以看到，这里的调用关系和 runtime-dom、runtime-core 的关系类似，compiler-dom 负责传入浏览器 Dom 相关的 API，实际编译的 baseCompile 是由 compiler-core 提供的。

复制代码

```
export function compile(
  template: string,
  options: CompilerOptions = {}
): CodegenResult {
  return baseCompile(
    template,
    extend({}, parserOptions, options, {
      nodeTransforms: [
        // ignore <script> and <tag>
        // this is not put inside DOMNodeTransforms because that list is used
        // by compiler-ssr to generate vnode fallback branches
        ignoreSideEffectTags,
        ...DOMNodeTransforms,
        ...(options.nodeTransforms || [])
      ],
      directiveTransforms: extend(
        {},
        DOMDirectiveTransforms,
        options.directiveTransforms || {}
      ),
      transformHoist: __BROWSER__ ? null : stringifyStatic
    })
  )
}
```

我们先来看看 compiler-dom 做了哪些额外的配置。

首先，parserOption 传入了 parse 的配置，通过 parserOption 传递的 isNativeTag 来区分 element 和 component。这里的实现也非常简单，把所有 html 的标签名存储在一个对象中，然后就可以很轻松地判断出 div 是浏览器自带的 element。

baseCompile 传递的其他参数 nodeTransforms 和 directiveTransforms，它们做的也是和上面代码类似的事。

复制代码

```
export const parserOptions: ParserOptions = {
  isVoidTag,
  isNativeTag: tag => isHTMLTag(tag) || isSVGTag(tag),
  isPreTag: tag => tag === 'pre',
  decodeEntities: __BROWSER__ ? decodeHtmlBrowser : decodeHtml,

  isBuiltInComponent: (tag: string): symbol | undefined => {
    if (isBuiltInType(tag, `Transition`)) {
      return TRANSITION
    } else if (isBuiltInType(tag, `TransitionGroup`)) {
      return TRANSITION_GROUP
    }
  },
  ...
}
const HTML_TAGS =
  'html,body,base,head,link,meta,style,title,address,article,aside,footer,' +
  'header,h1,h2,h3,h4,h5,h6,nav,section,div,dd,dl,dt,figcaption,' +
  'figure,picture,hr,img,li,main,ol,p,pre,ul,a,b,abbr,bdi,bdo,br,cite,code,' +
  'data,dfn,em,i,kbd,mark,q,rp,rt,ruby,s,samp,small,span,strong,sub,sup,' +
  'time,u,var,wbr,area,audio,map,track,video,embed,object,param,source,' +
  'canvas,script,noscript,del,ins,caption,col,colgroup,table,thead,tbody,td,'
  'th,tr,button,datalist,fieldset,form,input,label,legend,meter,optgroup,' +
  'option,output,progress,select,textarea,details,dialog,menu,' +
  'summary,template,blockquote,iframe,tfoot'
export const isHTMLTag = /*#__PURE__*/ makeMap(HTML_TAGS)
```

Vue 浏览器端编译的核心流程

然后，我们进入到 baseCompile 函数中，这就是 Vue 浏览器端编译的核心流程。

下面的代码中可以很清楚地看到，我们先通过 baseParse 把传递的 template 解析成 AST，然后通过 transform 函数对 AST 进行语义化分析，最后通过 generate 函数生成代码。

这个主要逻辑和我们写的迷你 compiler 基本一致，这些函数大概要做的事你也心中有数了。这里你也能体验到，亲手实现一个迷你版本对我们阅读源码很有帮助。

接下来，我们就进入到这几个函数之中去，看一下跟迷你 compiler 里的实现相比，我们到底做了哪些优化。

复制代码

```
export function baseCompile(
  template: string | RootNode,
  options: CompilerOptions = {}
): CodegenResult {
  const ast = isString(template) ? baseParse(template, options) : template
  const [nodeTransforms, directiveTransforms] =
    getBaseTransformPreset(prefixIdentifiers)

  transform(
    ast,
    extend({}, options, {
      prefixIdentifiers,
      nodeTransforms: [
        ...nodeTransforms,
        ...(options.nodeTransforms || []) // user transforms
      ],
      directiveTransforms: extend(
        {},
        directiveTransforms,
        options.directiveTransforms || {} // user transforms
      )
    })
  )
  return generate(
    ast,
    extend({}, options, {
      prefixIdentifiers
    })
  )
}

```

上一讲中我们体验了 Vue 的在线模板编译环境，可以在 console 中看到 Vue 解析得到的 AST。

如下图所示，可以看到这个 AST 比迷你版多了很多额外的属性。**loc 用来描述节点对应代码的信息，component 和 directive 用来记录代码中出现的组件和指令等等。**

然后我们进入到 baseParse 函数中，这里的 createParserContext 和 createRoot 用来生成上下文，其实就是创建了一个对象，保存当前 parse 函数中需要共享的数据和变量，最后调用 parseChildren。

children 内部开始判断 < 开头的标识符，判断开始还是闭合标签后，接着会生成一个 nodes 数组。其中，advanceBy 函数负责更新 context 中的 source 用来向前遍历代码，最终对不同的场景执行不同的函数。

复制代码

```
export function baseParse(
  content: string,
  options: ParserOptions = {}
): RootNode {
  const context = createParserContext(content, options)
  const start = getCursor(context)
  return createRoot(
    parseChildren(context, TextModes.DATA, []),
    getSelection(context, start)
  )
}
function parseChildren(
  context: ParserContext,
  mode: TextModes,
  ancestors: ElementNode[]
): TemplateChildNode[] {
  const parent = last(ancestors)
  // 依次生成node
  const nodes: TemplateChildNode[] = []
```

```
// 如果遍历没结束
while (!isEnd(context, mode, ancestors)) {

  const s = context.source
  let node: TemplateChildNode | TemplateChildNode[] | undefined = undefined

  if (mode === TextModes.DATA || mode === TextModes.RCDATA) {
    if (!context.inVPre && startsWith(s, context.options.delimiters[0])) {
      // 处理vue的变量标识符，两个大括号 '{{'
      node = parseInterpolation(context, mode)
    } else if (mode === TextModes.DATA && s[0] === '<') {
      // 处理<开头的代码，可能是<div>也有可能是</div> 或者<!的注释
      if (s.length === 1) {
        // 长度是1，只有一个< 有问题 报错
        emitError(context, ErrorCodes.EOF_BEFORE_TAG_NAME, 1)
      } else if (s[1] === '!') {
        // html注释
        if (startsWith(s, '<!--')) {
          node = parseComment(context)
        } else if (startsWith(s, '<!DOCTYPE')) {

          // DOCTYPE
          node = parseBogusComment(context)
        }
      } else if (s[1] === '/') {
         //</ 开头的标签，结束标签
        // https://html.spec.whatwg.org/multipage/parsing.html#end-tag-open-
        if (/[a-z]/i.test(s[2])) {
          emitError(context, ErrorCodes.X_INVALID_END_TAG)
          parseTag(context, TagType.End, parent)
          continue
        }
      } else if (/[a-z]/i.test(s[1])) {
        // 解析节点
        node = parseElement(context, ancestors)
        // 2.x <template> with no directive compat
        node = node.children
        }
      }
    }
  }
  if (!node) {
    // 文本
    node = parseText(context, mode)
  }
  // node树数组，遍历puish
  if (isArray(node)) {
    for (let i = 0; i < node.length; i++) {
      pushNode(nodes, node[i])
    }
  } else {
    pushNode(nodes, node)
```

```
72      }
73    }
74
75    return removedWhitespace ? nodes.filter(Boolean) : nodes
76
```

parseInterpolation 和 parseText 函数的逻辑比较简单。parseInterpolation 负责识别变量的分隔符 {{ 和}} ，然后通过 parseTextData 获取变量的值，并且通过 innerStart 和 innerEnd 去记录插值的位置；parseText 负责处理模板中的普通文本，主要是把文本包裹成 AST 对象。

接着我们看看处理节点的 parseElement 函数都做了什么。首先要判断 pre 和 v-pre 标签，然后通过 isVoidTag 判断标签是否是自闭合标签，这个函数是从 compiler-dom 中传来的，之后会递归调用 parseChildren，接着再解析开始标签、解析子节点，最后解析结束标签。

复制代码

```
const VOID_TAGS =
  'area,base,br,col,embed,hr,img,input,link,meta,param,source,track,wbr'

export const isVoidTag = /*#__PURE__*/ makeMap(VOID_TAGS)
function parseElement(
  context: ParserContext,
  ancestors: ElementNode[]
): ElementNode | undefined {
  // Start tag.
  // 是不是pre标签和v-pre标签
  const wasInPre = context.inPre
  const wasInVPre = context.inVPre
  const parent = last(ancestors)
  // 解析标签节点
  const element = parseTag(context, TagType.Start, parent)
  const isPreBoundary = context.inPre && !wasInPre
  const isVPreBoundary = context.inVPre && !wasInVPre

  if (element.isSelfClosing || context.options.isVoidTag(element.tag)) {
    // #4030 self-closing <pre> tag
    if (isPreBoundary) {
      context.inPre = false
    }
    if (isVPreBoundary) {
      context.inVPre = false
    }
    return element
  }

```

```
  // Children.
  ancestors.push(element)
  const mode = context.options.getTextMode(element, parent)
  const children = parseChildren(context, mode, ancestors)
  ancestors.pop()
  element.children = children

  // End tag.
  if (startsWithEndTagOpen(context.source, element.tag)) {
    parseTag(context, TagType.End, parent)
  } else {
    emitError(context, ErrorCodes.X_MISSING_END_TAG, 0, element.loc.start)
    if (context.source.length === 0 && element.tag.toLowerCase() === 'script')
      const first = children[0]
      if (first && startsWith(first.loc.source, '<!--')) {
        emitError(context, ErrorCodes.EOF_IN_SCRIPT_HTML_COMMENT_LIKE_TEXT)
      }
    }
  }

  element.loc = getSelection(context, element.loc.start)

  if (isPreBoundary) {
    context.inPre = false
  }
  if (isVPreBoundary) {
    context.inVPre = false
  }
  return element
}
```

最后，我们来看下解析节点的 parseTag 函数的逻辑，匹配文本标签结束的位置后，先通过 parseAttributes 函数处理属性，然后对 pre 和 v-pre 标签进行检查，最后通过 isComponent 函数判断是否为组件。

isComponent 内部会通过 compiler-dom 传递的 isNativeTag 来辅助判断结果，最终返回一个描述节点的对象，包含当前节点所有解析之后的信息，tag 表示标签名，children 表示子节点的数组，具体代码我放在了后面。

复制代码

```
function parseTag(
  context: ParserContext,
  type: TagType,
  parent: ElementNode | undefined
): ElementNode | undefined {

```

```
  // Tag open.
  const start = getCursor(context)
  //匹配标签结束的位置
  const match = /^<\/?([a-z][^\t\r\n\f />]*)/i.exec(context.source)!
  const tag = match[1]
  const ns = context.options.getNamespace(tag, parent)
  // 向前遍历代码
  advanceBy(context, match[0].length)
  advanceSpaces(context)

  // save current state in case we need to re-parse attributes with v-pre
  const cursor = getCursor(context)
  const currentSource = context.source

  // check <pre> tag
  if (context.options.isPreTag(tag)) {
    context.inPre = true
  }
  // Attributes.
  // 解析属性
  let props = parseAttributes(context, type)
  // check v-pre
  if (){...}
  // Tag close.
  let isSelfClosing = false
  if (type === TagType.End) {
    return
  }

  let tagType = ElementTypes.ELEMENT
  if (!context.inVPre) {
    if (tag === 'slot') {
      tagType = ElementTypes.SLOT
    } else if (tag === 'template') {
      if (
        props.some(
          p =>
            p.type === NodeTypes.DIRECTIVE && isSpecialTemplateDirective(p.nam
        )
      ) {
        tagType = ElementTypes.TEMPLATE
      }
    } else if (isComponent(tag, props, context)) {
      tagType = ElementTypes.COMPONENT
    }
  }

  return {
    type: NodeTypes.ELEMENT,
    ns,
    tag,
    tagType,
```

```
    props,
    isSelfClosing,
    children: [],
    loc: getSelection(context, start),
    codegenNode: undefined // to be created during transform phase
  }
}

```

parse 函数生成 AST 之后，我们就有了一个完整描述 template 的对象，它包含了 template 中所有的信息。

AST 的语义化分析

下一步我们要对 AST 进行语义化的分析。transform 函数的执行流程分支很多，**核心的逻辑就是识别一个个的 Vue 的语法，并且进行编译器的优化，我们经常提到的静态标记就是这一步完成的。**

我们进入到 transform 函数中，可以看到，内部通过 createTransformContext 创建上下文对象，这个对象包含当前分析的属性配置，包括是否 ssr，是否静态提升还有工具函数等等，这个对象的属性你可以在 GitHub上看到。

复制代码

```


export function transform(root: RootNode, options: TransformOptions) {
  const context = createTransformContext(root, options)
  traverseNode(root, context)
  if (options.hoistStatic) {
    hoistStatic(root, context)
  }
  if (!options.ssr) {
    createRootCodegen(root, context)
  }
  // finalize meta information
  root.helpers = [...context.helpers.keys()]
  root.components = [...context.components]
  root.directives = [...context.directives]
  root.imports = context.imports
  root.hoists = context.hoists
  root.temps = context.temps
  root.cached = context.cached

  if (__COMPAT__) {
```

```
    root.filters = [...context.filters!]
  }
}

```

然后通过 traverseNode 即可编译 AST 所有的节点。核心的转换流程是在遍历中实现，内部使用 switch 判断 node.type 执行不同的处理逻辑。比如如果是 Interpolation，就需要在 helper 中导入 toDisplayString 工具函数，这个迷你版本中我们也实现过。

复制代码

```

export function traverseNode(
  node: RootNode | TemplateChildNode,
  context: TransformContext
) {
  context.currentNode = node
  // apply transform plugins
  const { nodeTransforms } = context
  const exitFns = []
  for (let i = 0; i < nodeTransforms.length; i++) {
    // 处理exitFns
  }
  swtch (node.type) {
    case NodeTypes.COMMENT:
      if (!context.ssr) {
        context.helper(CREATE_COMMENT)
      }
      break
    case NodeTypes.INTERPOLATION:
      if (!context.ssr) {
        context.helper(TO_DISPLAY_STRING)
      }
      break
    case NodeTypes.IF:
      for (let i = 0; i < node.branches.length; i++) {
        traverseNode(node.branches[i], context)
      }
      break
    case NodeTypes.IF_BRANCH:
    case NodeTypes.FOR:
    case NodeTypes.ELEMENT:
    case NodeTypes.ROOT:
      traverseChildren(node, context)
      break
  }

  // exit transforms
  context.currentNode = node
```

```
  let i = exitFns.length
  while (i--) {
    exitFns[i]()
  }
}

```

transform 中还会调用 transformElement 来转换节点，用来处理 props 和 children 的静态标记，transformText 用来转换文本，这里的代码比较简单， 你可以自行在Github上查阅。

transform 函数参数中的 nodeTransforms 和 directiveTransforms 传递了 Vue 中 template 语法的配置，这个两个函数由 getBaseTransformPreset 返回。

下面的代码中，transformIf 和 transformFor 函数式解析 Vue 中 v-if 和 v-for 的语法转换，transformOn 和 transformModel 是解析 v-on 和 v-model 的语法解析，这里我们只关注 v- 开头的语法。

复制代码

```


export function getBaseTransformPreset(
  prefixIdentifiers?: boolean
): TransformPreset {
  return [
    [
      transformOnce,
      transformIf,
      transformMemo,
      transformFor,
      ...(__COMPAT__ ? [transformFilter] : []),
      ...(!__BROWSER__ && prefixIdentifiers
        ? [
            // order is important
            trackVForSlotScopes,
            transformExpression
          ]
        : __BROWSER__ && __DEV__
        ? [transformExpression]
        : []),
      transformSlotOutlet,
      transformElement,
      trackSlotScopes,
      transformText
```

```
    ],
    {
      on: transformOn,
      bind: transformBind,
      model: transformModel
    }
  ]
}

```

然后我们再来看看 transformIf 的函数实现。首先判断 v-if、v-else 和 v-else-if 属性，内部通过 createCodegenNodeForBranch 来创建条件分支，在 AST 中标记当前 v-if 的处理逻辑。这段逻辑标记结束后，在 generate 中就会把 v-if 标签和后面的 v-else 标签解析成三元表达式。

复制代码

```
export const transformIf = createStructuralDirectiveTransform(
  /^(if|else|else-if)$/,
  (node, dir, context) => {
    return processIf(node, dir, context, (ifNode, branch, isRoot) => {
      const siblings = context.parent!.children
      let i = siblings.indexOf(ifNode)
      let key = 0
      while (i-- >= 0) {
        const sibling = siblings[i]
        if (sibling && sibling.type === NodeTypes.IF) {
          key += sibling.branches.length
        }
      }
      return () => {
        if (isRoot) {
          ifNode.codegenNode = createCodegenNodeForBranch(
            branch,
            key,
            context
          ) as IfConditionalExpression
        } else {
          // attach this branch's codegen node to the v-if root.
          const parentCondition = getParentCondition(ifNode.codegenNode!)
          parentCondition.alternate = createCodegenNodeForBranch(
            branch,
            key + ifNode.branches.length - 1,
            context
          )
        }
      }
    })
  }
```

```
33 )
```

transform 对 AST 分析结束之后，我们就得到了一个优化后的 AST 对象，最后我们需要调用 generate 函数最终生成 render 函数。

template 到 render 函数的转化

结合下面的代码我们可以看到，generate 首先通过 createCodegenContext 创建上下文对象，然后通过 genModulePreamble 生成预先定义好的代码模板，然后生成 render 函数，最后生成创建虚拟 DOM 的表达式。

复制代码

```
export function generate(
  ast,
  options
): CodegenResult {
  const context = createCodegenContext(ast, options)
  const {
    mode,
    push,
    prefixIdentifiers,
    indent,
    deindent,
    newline,
    scopeId,
    ssr
  } = context

  if (!__BROWSER__ && mode === 'module') {
    // 预设代码，module风格 就是import语句
    genModulePreamble(ast, preambleContext, genScopeId, isSetupInlined)
  } else {
    // 预设代码，函数风格 就是import语句
    genFunctionPreamble(ast, preambleContext)
  }
  // render还是ssrRender
  const functionName = ssr ? `ssrRender` : `render`
  const args = ssr ? ['_ctx', '_push', '_parent', '_attrs'] : ['_ctx', '_cache
  if (!__BROWSER__ && options.bindingMetadata && !options.inline) {
    // binding optimization args
    args.push('$props', '$setup', '$data', '$options')
  }
  const signature =
    !__BROWSER__ && options.isTS
      ? args.map(arg => `${arg}: any`).join(',')
      : args.join(', ')
```

```
35
36    if (isSetupInlined) {
37      push(`(${signature}) => {`)
38    } else {
39      push(`function ${functionName}(${signature}) {`)
40    }
41    indent()
42
43    // 组件，指令声明代码
44    if (ast.components.length) {
45      genAssets(ast.components, 'component', context)
46      if (ast.directives.length || ast.temps > 0) {
47        newline()
48      }
49    }
50    if (ast.components.length || ast.directives.length || ast.temps) {
51      push(`\n`)
52      newline()
53    }
54
55    if (ast.codegenNode) {
56      genNode(ast.codegenNode, context)
57    } else {
58      push(`null`)
59    }
60
61    if (useWithBlock) {
62      deindent()
63      push(`}`)
64    }
65
66    deindent()
67    push(`}`)
68
69    return {
70      ast,
71      code: context.code,
72      preamble: isSetupInlined ? preambleContext.code : ``,
73      // SourceMapGenerator does have toJSON() method but it's not in the types
74      map: context.map ? (context.map as any).toJSON() : undefined
75    }
76  }
77
```

我们来看下关键的步骤，genModulePreamble 函数生成 import 风格的代码，这也是我们迷你版本中的功能：通过遍历 helpers，生成 import 字符串，这对应了代码的第二行。

复制代码

```
// 生成这个
// import { toDisplayString as _toDisplayString, createElementVNode as _create

function genModulePreamble(
  ast: RootNode,
  context: CodegenContext,
  genScopeId: boolean,
  inline?: boolean
) {

  if (genScopeId && ast.hoists.length) {
    ast.helpers.push(PUSH_SCOPE_ID, POP_SCOPE_ID)
  }
  // generate import statements for helpers
  if (ast.helpers.length) {
      push(
        `import { ${ast.helpers
          .map(s => `${helperNameMap[s]} as _${helperNameMap[s]}`)
          .join(', ')} } from ${JSON.stringify(runtimeModuleName)}\n`
      )
    }
  }
  ...
}
```

接下来的步骤就是生成渲染函数 render 和 component 的代码，最后通过 genNode 生成创建虚拟的代码，执行 switch 语句生成不同的代码，一共有十几种情况，这里就不一一赘述了。我们可以回顾上一讲中迷你代码的逻辑，总之针对变量，标签，v-if 和 v-for 都有不同的代码生成逻辑，最终才实现了 template 到 render 函数的转化。

复制代码

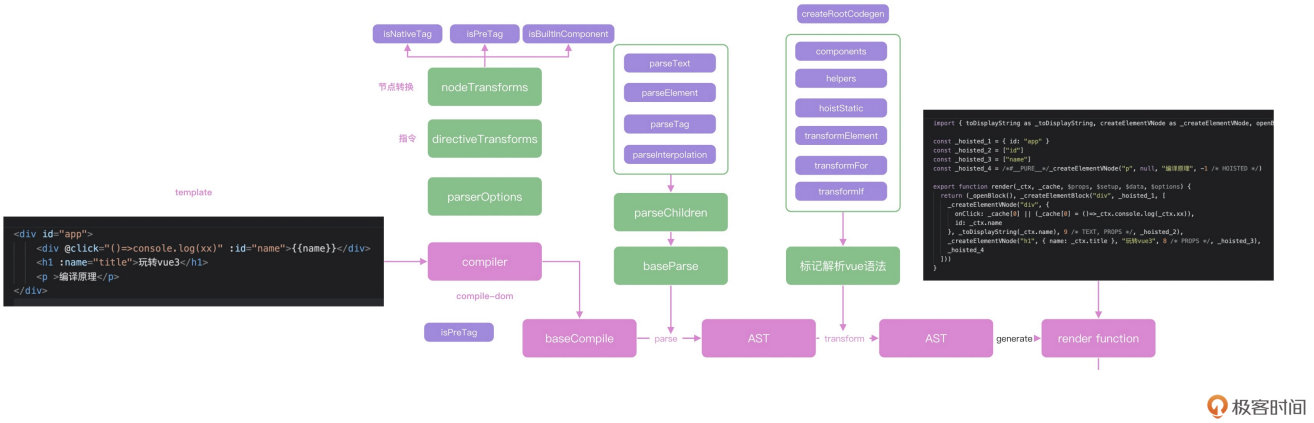
```
function genNode(node: CodegenNode | symbol | string, context: CodegenContext)
  if (isString(node)) {
    context.push(node)
    return
  }
  if (isSymbol(node)) {
    context.push(context.helper(node))
    return
  }
  switch (node.type) {
    case NodeTypes.ELEMENT:
    case NodeTypes.IF:
    case NodeTypes.FOR:
      genNode(node.codegenNode!, context)
      break
    case NodeTypes.TEXT:
```

```
      genText(node, context)
      break
    case NodeTypes.SIMPLE_EXPRESSION:
      genExpression(node, context)
      break
    case NodeTypes.INTERPOLATION:
      genInterpolation(node, context)
      break
    case NodeTypes.TEXT_CALL:
      genNode(node.codegenNode, context)
      break
    case NodeTypes.COMPOUND_EXPRESSION:
      genCompoundExpression(node, context)
      break
    case NodeTypes.COMMENT:
      genComment(node, context)
      break
    case NodeTypes.VNODE_CALL:
      genVNodeCall(node, context)
      break

    case NodeTypes.JS_CALL_EXPRESSION:
      genCallExpression(node, context)
      break
    case NodeTypes.JS_OBJECT_EXPRESSION:
      genObjectExpression(node, context)
      break
    case NodeTypes.JS_ARRAY_EXPRESSION:
      genArrayExpression(node, context)
      break
    case NodeTypes.JS_FUNCTION_EXPRESSION:
      genFunctionExpression(node, context)
      break
    case NodeTypes.JS_CONDITIONAL_EXPRESSION:
      genConditionalExpression(node, context)
      break
    case NodeTypes.JS_CACHE_EXPRESSION:
      genCacheExpression(node, context)
      break
    case NodeTypes.JS_BLOCK_STATEMENT:
      genNodeList(node.body, context, true, false)
      break

    /* istanbul ignore next */
    case NodeTypes.IF_BRANCH:
      // noop
      break

  }
}
```

总结

今天的内容到这就讲完了，我给你总结一下今天讲到的内容吧。

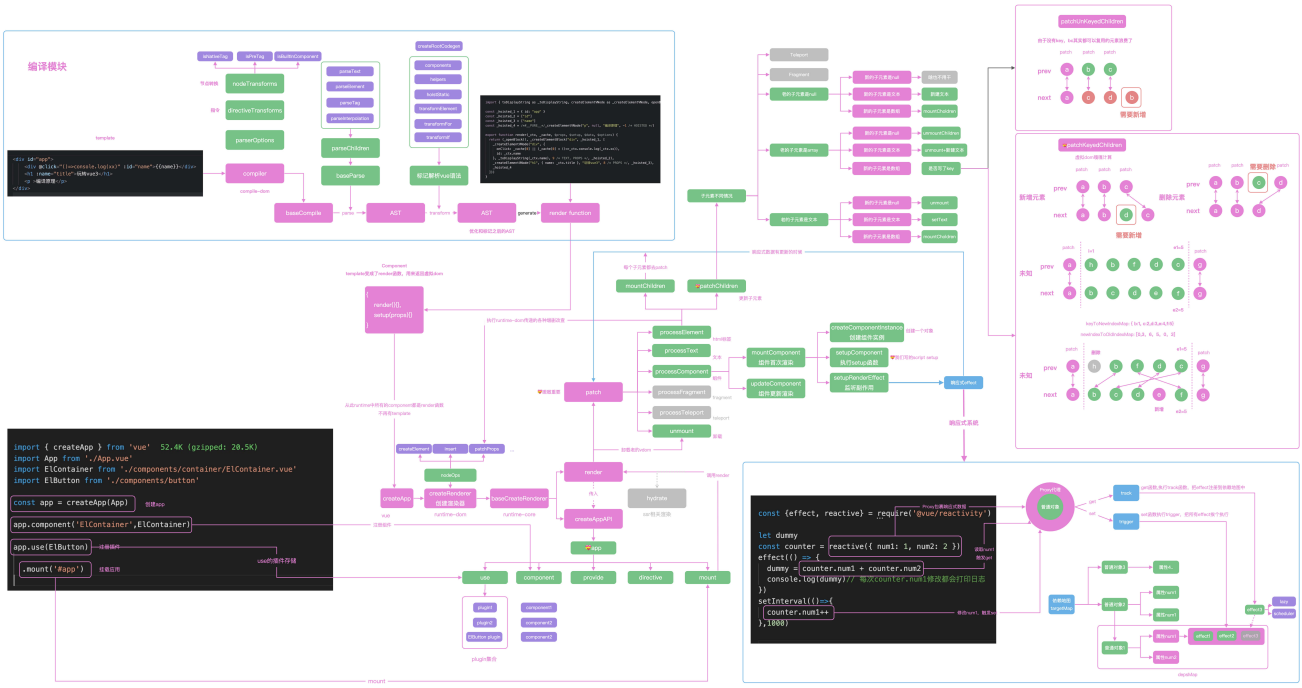
今天我们一起分析了 Vue 中的 compiler 执行全流程，有了上一讲编译入门知识的基础之后，今天的 parse，transform 和 generate 模块就是在上一讲的基础之上，更加全面地实现代码的编译和转化。



上面的流程图中，我们代码中的 template 是通过 compiler 函数进行编译转换，compiler 内部调用了 compiler-core 中的 baseCompile 函数，并且传递了浏览器平台的转换逻辑。

比如 isNativeTag 等函数，baseCompie 函数中首先通过 baseParse 函数把 template 处理成为 AST，并且由 transform 函数进行标记优化，transfom 内部的 transformIf，transformOn 等函数会对 Vue 中的语法进行标记，这样在 generate 函数中就可以使用优化后的 AST 去生成最终的 render 函数。

最终，render 函数会和我们写的 setup 函数一起组成组件对象，交给页面进行渲染。后面我特意为你绘制了一幅 Vue 全流程的架构图，你可以保存下来随时查阅。



极客时间

Vue 源码中的编译优化也是 Vue 框架的亮点之一，我们自己也要思考编译器优化的机制，可以提高浏览器运行时的性能，我们项目中该如何借鉴这种思路呢？下一讲我会详细剖析编译原理在实战里的应用，敬请期待。

思考题

最后留一个思考题，transform 函数中针对 Vue 中的语法有很多的函数处理，比如 transformIf 会把 v-if 指令编译成为一个三元表达式，请你从其余的函数选一个在评论区分享 transform 处理的结果吧。欢迎在评论区分享你的答案，我们下一讲再见！

分享给需要的人，Ta订阅后你可得 20 元现金奖励

生成海报并分享

赞 3 提建议

© 版权归极客邦科技所有，未经许可不得传播售卖。 页面已增加防盗追踪，如有侵权极客邦将依法追究其法律责任。

上一篇 32 | 编译原理（上）：手写一个迷你Vue 3 Compiler的入门原理

下一篇 34 | 编译原理（下）：编译原理给我们带来了什么？

更多课程推荐

精选留言（1）

写留言

斜月浮云

2022-01-06

全流程汇总图好评哦👍